Bachelorarbeit

Implementing Monte Carlo Tree Search For The Vienna Game AI Library

Verfasser

Bleon Jupa

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2026

Studienkennzahl lt. Studienblatt:	UA 033 521
Fachrichtung:	Bachelorstudium Informatik
Betreuer:	Univ.Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Acknowledgments

I would like to express my gratitude first and foremost to my supervisor Univ.Prof. Dipl.-Ing. Dr. Helmut Hlavacs who has helped me navigate ideas and find solutions for problems that arose during the research and development process. His understanding and tolerance for me needing more time or struggling with certain parts made working on this thesis a lot less stressful and more manageable.

I would also like to thank my parents for their support, as well as my friends, Lamies Abbas, Kalian Danzer, Andy Liu and Matthäus Wininger, all of whom made studying computer science very enjoyable.

Abstract

This thesis presents a Monte Carlo tree search implementation for the Vienna Game AI Library for game developers to use effectively, without requiring heuristics and neither producing any substantial overhead to be integrated into projects. Furthermore, the thesis explores the performance of Monte Carlo tree search across games with different complexities, and analyses in particular if there is a discrepancy in playing strength differences with scaling thread counts.

In order to address the research question, test runs were being conducted in the games Tic-Tac-Toe, Connect Four and Hex. Afterwards, the statistics obtained were first analysed per type of game, and then a comparison was made between all three types of games. The results show that a discrepancy, as well as a widening performance gap between single-threaded MCTS and multi-threaded MCTS with a high thread count, can be found in games with higher branching factors.

Contents

1 Introduction **5**

2 Related Work **6**

3 Technical Foundations **10**
3.1 Monte Carlo Method 10
3.2 Monte Carlo Tree Search 10
3.3 Upper Confidence Bounds Applied To Trees 14
3.4 Parallelization 15
3.5 Vienna Game AI Library 16

4 Implementing MCTS **16**
4.1 Implementation Design 16
4.2 Class Action 16
4.3 Class MCTSState 17
4.4 Class MCTSNode 18
4.5 Class MCTS 19
4.6 Examplatory Use 23

5 Evaluation **25**
5.1 Experiment Results 26
5.2 Discussion 28

6 Conclusion **29**

1 Introduction

With its roots ranging back to the 1940s [17], and formally being proposed in 2006 [8], Monte Carlo tree search (MCTS) started as a strong decision-making algorithm for solving sequential computer games, most notably Go. Unlike other popular decision-making algorithms at the time, like minimax with alpha-beta pruning, MCTS does not rely on heuristics. The algorithm is based on building a search tree by performing many playouts, also sometimes called rollouts, consisting of four phases (selection phase, expansion phase, simulation phase and backpropagation phase). In each playout the search tree is expanded and a game is simulated by performing random actions until the game has terminated. The more playouts are being conducted the more statistics MCTS can gather and decide which next move is the most promising one.

With the introduction of Upper Confidence Bounds applied to trees (UCT) [14] the algorithm managed to gain popularity in the Go playing scene. Parallelization strategies, such as leaf parallelization [4], tree parallelization [5] and root parallelization [6], managed to further increase the performance of MCTS and were of particular interest in applications beyond games. Not only was MCTS quickly considered a success for artificial intelligence (AI), but the combination of MCTS with neural networks, such as in the case of AlphaGo [22], proved promising as a field of future research regarding machine learning (ML). While MCTS is not typically associated with ML, but rather classical AI search problems, it enabled MCTS to widen its range of applications beyond board games. In recent years MCTS has seen use in all kinds of domains, like scheduling forest harvesting [19], scheduling patrols at the LAX airport [13], robotics [20] and as a model in the prevention of poaching of endangered species [10].

The goal of this thesis is to implement a simple yet well-performing MCTS implementation for the Vienna Game AI Library (VGAI) [15] for game developers to use, while trying to minimize the implementation overhead on the game developers' side. The implementation design should be based on insight gained during the research of existing MCTS variants, choosing an appropriate selection policy and a suitable parallelization strategy. Building on these criteria the following research question arose:

- How does the playing strength of MCTS without domain knowledge scale with the number of threads across games with different complexities?

In order to answer the research question three games (Tic-Tac-Toe, Connect Four and Hex) have been implemented. These games differ in their branching factors by a factor of approximately three to ten, giving us an adequate range for evaluation. Our results show a noticeable discrepancy in performance of MCTS with a scaling

thread count in games with large branching factors, while games with low branching factors do not indicate such.

2 Related Work

Since its introduction as a sequential decision-making algorithm for board games, MCTS has been highly studied and over time seen more and more applications outside of games. This section reviews the modifications the algorithm has been subjected to, aimed at improving, but not limited to, decision-making quality, computation time or scalability.

By introducing Upper Confidence Bounds applied to Trees [14] (UCT) the selection policy could be properly balanced between exploiting the best actions known so far and exploring not well-tested actions. By applying traditional random sampling and UCT, the MCTS search tree is explored asymmetrically, with promising actions being explored more deeply. This resulted in a considerably faster convergence to the optimal action. However, shortest path problems (SSP) like the sailing problem need further future analysis, since this domain could not be proven to benefit from UCT.

In order to handle large branching factors, Coulum [8] proposed progressive widening. This technique adds a new child node to the search tree gradually based on visit counts. By doing so, scalability improved in high-dimensional action spaces. Using this technique, a computer-Go program managed to convincingly win against a state-of-the-art Monte-Carlo Go-playing program in a 100-game-series. While progressive widening does improve performance in domains with large branching factors, the reliance on random simulations remains a limtiing factor as well as possibility for future research.

Couëtoux et al. [7] extended progressive widening to continuous state and action spaces by introducing double progressive widening. In continuous action spaces there are infinitely many actions, for example choosing a real value between two numbers, thus intelligent sampling is required. Double progressive widening extends progressive widening to handle the expansion of both action space and the resulting states by limiting the number of actions that can be explored in a node. Their experiments showed an improved performance in continuous settings versus UCT or progressive widening. Nevertheless, performance in very high-dimensional continuous domains remains untested.

Gelly and Silver [11] proposed Rapid Action Value Estimation (RAVE) to accelerate the convergence towards the optimal play. In standard MCTS, only actions that lie directly in between the root node and the leaf node get updated after a simulation, while RAVE reuses information gathered during a simulation to store statistics for actions chosen during the simulation. When it is time to expand a node,

the algorithm has already estimates about the value of actions instead of having to rely on random expansion. While RAVE showed faster convergence compared to standard UCT, optimal weighting guarantees between UCT and RAVE have been left open for future research.

Although prior research has been conducted to improve decision-making quality, Chaslot et al. [6] explored parallelization strategies to accelerate search time. They presented three types of parallization: Leaf parallelization, tree parallelization, and root paralellization. Results showed a higher speed-up while retaining a similar playing strength in games like Go. Nonetheless, open questions remained regarding scalability and synchronisation strategies.

Cowling et al. [9] explored application methods of MCTS in games where information is hidden or uncertain. They presented three information set MCTS (ISMCTS) algorithms where, instead of building a search tree consisting of states, it contains information sets as nodes. By using ISMCTS they achieved a stronger performance in card games like Knockout Whist and Dou Di Zhu than previous determinized MCTS variants. Non-locality and opponent modelling problems, however, have been considered out of the scope of their work and left out for future research.

AlphaGo [22] was the first system to defeat a professional Go player without handicap. It combined MCTS with deep neural networks, first trained through supervised learning on human games and then by reinforcement learning through playing games by itself. Even though it proved capable to compete in a professionaö setting, it suffers from needing heavy domain knowledge and pretraining.

Silver et al. [24] introduced AlphaGo Zero, improving AlphaGo by removing human data from training, relying entirely on self-play. It managed to surpass AlphaGo in playing strength after only a few days of training. Nonetheless, it required high computational power, making it unsuitable for domains where such budgets are not available. Additionally, exact training details, like hyperparameters or used infrastructure were not revealed.

With the introduction of AlphaZero [23] the AlphaGo system was expanded to be applicable in other games than just Go by changing the hard-coded Go-specific code to a game-agnostic interface. Using the same neural network, it achieved superhuman performance in multiple board games like Go, Chess and Shogi.

Schrittwieser et al. [21] proposed MuZero to provide a MCTS system that learns to play games without being told the game rules. Instead of reconstructing the full environment in a game after every move, it only learns a model of the game environment and makes assumptions about what is needed for decision-making. Their results showed a strong performance in board games. Like AlphaZero, however, it needs high computational power and applications outside of board games remain untested.

In order to tackle the problem of combinatorial action spaces (CAS), where each action consists of multiple independent choices, Ontañión [20] proposed Naive Monte Carlo tree search (NaMCTS). NaMCTS assumes that the value of a combined action can be approximated through the values of each individual action, making it more scalable for domains like RTS games, where standard MCTS and UCT struggle to deliver a strong performance. He evaluated the playing strength of NaMCTS, standard MCTS, UCT and other algorithms in microRTS, a simplified RTS game for AI research. NaMCTS yielded comparable playing strength compared to the other algorithms in scenarios with small branching factors, and outperformed them as the branching factors grew.

Uriarte et al. [27] proposed single believe state generation in MCTS to represent unknown information, such as fog-of-war in RTS games, instead of maintaining multiple different states for all possibilities. By using stochastic likelihood and domain knowledge the most likely game state is estimated for the algorithm. Their results show the method achieves a comparable playing strength, as opposed to other MCTS systems that use multiple state generation for hidden information, while requiring less computational power. However, it has been left open for future research whether similar results can be achieved when scaled to large-world RTS games.

Neto et al. [19] researched the application of MCTS to the scheduling of forest harvesting by modifying the algorithm to consider trade-offs between maximizing profit and minimizing environmental impact, rather than optimizing a single objective. Their results show that multi-objective MCTS is able to find a better balanced solution than single-objective MCTS alternatives.

Hennes et al. [12] studied the use of MCTS to plan interplanetary space missions. They modeled the trajectory of a spacecraft as a sequence of maneuvers, with the aim to efficiently find good sequences of gravity assists without requiring heuristics. Although, in their experiments, MCTS not only managed to find sequences that were previously known to be good, but also new trajectory solutions. It has been noted that potential future work could lie in the addition of a second objective to score trajectories, i.e. total flight time, in order to enable the search for solutions with more specifications.

Best et al. [3] proposed Dec-MCTS, a decentralised MCTS system to control multiple robots in perception tasks. Each robot builds its own search tree and periodically communicates a compressed form of the search tree with the other robots. Although the algorithm performs favourably compared to centralized MCTS, even with significantly lower and less reliable communication, their results indicate redundant information sharing, leaving room for improvement in communication.

Clary et al. [20] introduced a variant of MCTS for high-level legged locomotion planning, named Monte-Carlo Discrepancy Search (MCDS), by combining a self-

stable blind-walking controller with a Monte-Carlo planner that guides it through and around obstacles. In their experiments MCDS outperformed other MCTS variants in both deterministic and stochastic environments, achieving a 100% success rate on their planar domain and 95% success rate on their full-order domain. Potential future work lies in improving the treatment of stochasticity, in order to ensure applications in broader domains where the controllers may be less self-stable.

Jain et al. [13] explored the use of MCTS to generate randomized patrols for the LAX airport, while ensuring different weights correspond to the different importance of targets. They report high-quality schedules being presented that were previously not able to be solved in realistic time-frames. Nonetheless, heavy use of domain knowledge is needed to guarantee optimal weights in the decision-making process.

An et al. [2] proposed PROTECT, a system that uses MCTS to schedule patrols in the port of Boston. It assumes security challenges as an attacker-defender model and, unlike typical game-theoretic models, does not speculate that the attacker behaves perfectly rational. In their experiments the United States Coast Guard (USCG) found the system to be a success compared to previous, both human-generated and computer-based, patrol scheduling systems.

Fang et al. [10] introduced Green Security Games (GSG), a game model for green security domains, such as the poaching of endangered species. The model uses MCTS to explore promosing patrol strategies where poachers and rangers take actions in alternating turns. Contrary to widely established systems, the poachers and rangers explore multiple actions of their action sets per iteration to simulate a mixed strategy. While in their experiments GSG performed quite well, testing in a real-world environment was left open for future research.

In their study, Wijaya et al. [28] present demand-side management (DSM) as a solution to manage peak electricity demand. DSM uses a dynamic pricing model which tries to tackle the herding effect where, once cheaper elecricity prices are offered at non-peak hours, a large portion of the consumers shift towards the cheaper price hours, creating new peak hours. Additionally, they rewarded so-called lazy solutions higher than those that require more effort, since the changed made to the grid should be kept as low as possible. In their simulations they managed to reduce peak hours while not necessarily causing different peak hours.

In their research Lubosch et al. [16] propose a new method to improve industrial scheduling. They combined MCTS with machine learning to reduce reliance on domain knowledge. Their findings show schedules that were more resilient to unexpected machine breakdowns than existing solutions can be achieved through random sampling combined with machine learning. How their system performs in larger suppy-chains was not addressed.

In order to improve the dispatching of autonomous transportation systems, such as in mining operations or in warehouses, Tomy et al. [26] present a variation

of MCTS by incorporating constraints, like penalties or violations, directly in the rewards of the search. Their results show an increase in task output in all four types of constraints they experimented on.

3 Technical Foundations

This section aims to provide the technical foundations required for the implementation of a Monte Carlo tree search algorithm. It explains theoretical concepts such as the core of the algorithm, covers modifications made to improve the convergence towards the optimal action, explores the most popular parallelization methods and covers the Vienna Game AI Library, where the algorithm was implemented. The emphasis is on explaining formulas and datasets presented by other researchers to form a bedrock for the next section.

3.1 Monte Carlo Method

In order to approach deterministic problems, which were difficult or impossible to solve with other methods, the Monte Carlo method [17] was developed by Stanislav Ulam while working on nuclear weapons projects at the Los Alamos National Laboratory in 1949. Instead of computing probablilities directly, the method estimates the solution by running many simulations and averaging the outcomes of the simulations.

Bruce Abramson [1] then, in 1987, combined the minimax search with an evaluation model which uses random playouts until the end, rather than a static evaluation model at the leaves, resulting in an early form of what would later influence Monte Carlo-based game search. Using this approach, sampling-based evaluation was able to outperform other evaluation functions of the time.

3.2 Monte Carlo Tree Search

In 2006 Coulom presented MCTS, a decision-making algorithm, by combining the Monte Carlo method with tree search for games. The main focus of MCTS lies on the evaluation of the most promising actions by building a search tree through continuous iterations, consisting of random simulations from leaf nodes and evaluating the outcomes, instead of using heuristics like other algorithms at the time. In the search tree, game states are depicted as nodes, with the lines connecting the nodes depicting the action it takes to go from one state to the other.

A key aspect of the algorithm is that the more iterations are conducted, the better the convergence towards the optimal action, since more time is being assigned to actions that have often times resulted in wins.

Each MCTS iteration consists of four phases:

- Selection phase: As the first phase of an iteration, the selection phase starts from the root node and, at each level, selects one of the child nodes from the current level until a node is reached that is either a terminal node, where the game has ended and no successive actions can be taken, or a leaf node. Leaf nodes are nodes that have at least one unexplored action that can be added as a new node to the tree. Figure 1 illustrates this by having the direct path from the root node to the selected leaf node coloured in magenta.

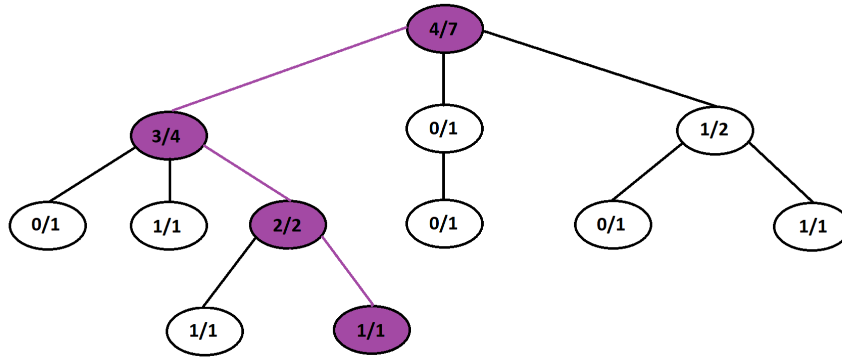

Figure 1: MCTS selection phase

- Expansion phase: After a leaf node has been reached at least one of its unexplored actions will be taken and the new game state added to the search tree as a node. Some implementations add only one node during this phase, while others add all possible nodes at once and choose one node at random to conduct the next phase from. The former method is generally more widely used, due to lower memory usage and better control of exploration. In figure 2 the newly added node is highlighted in magenta.

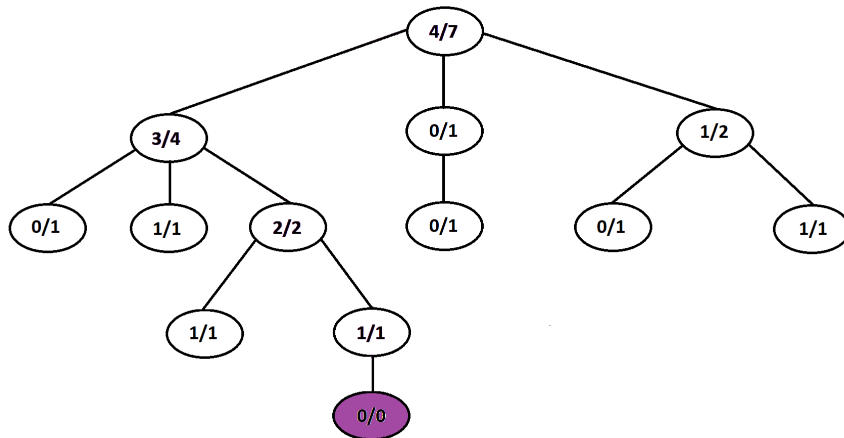

Figure 2: MCTS expansion phase

- Simulation phase: Once a new node has been added to the search tree the rest of the game is simulated by having random actions taken until the game reaches a terminal state. This is the "Monte Carlo" part of the algorithm. Any action taken during simulations, sometimes also called playouts or rollouts, will not be added as a new node to the tree. This phase is also computationally the most expensive phase of the algorithm. Figure 3 depicts this with a wavy line sparking from the expanded node towards the outcome of the simulation.

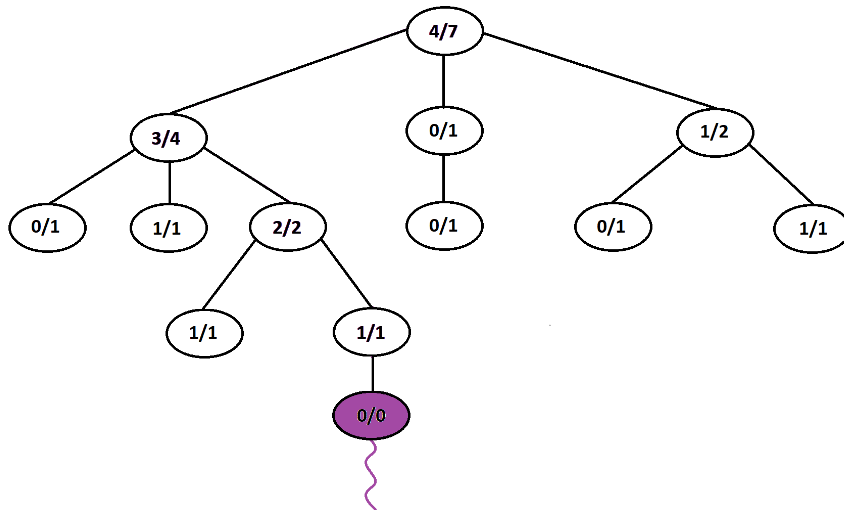

Figure 3: MCTS simulation phase

- Backpropagation phase: The last phase of the algorithm takes the outcome, often also called reward, of the simulation phase and updates the statistics of all nodes from the expanded node upwards up until the root node. Each node's visit count is incremented at all times, while the score depends on the weighting of the outcome. In some games the score can be either a win, resulting in simply incrementing the win count, or a loss, resulting in not increasing the win count, while in other games one win may be worth more than another and as such the win count is swapped with a total score for easier tracking. The latter method is often times used in implementations which incorporate heuristics. To illustrate this phase figure 4 shows the updated win and visit counts of the nodes in the search tree in magenta. Previously, the win and visit counts from the root node towards the expanded node were 4/7, 3/4, 2/2, 1/1, and 0/0. After the results are backpropagated, the updated win and visit counts show 5/8, 4/5, 3/3, 2/2, and 1/1, respectively.

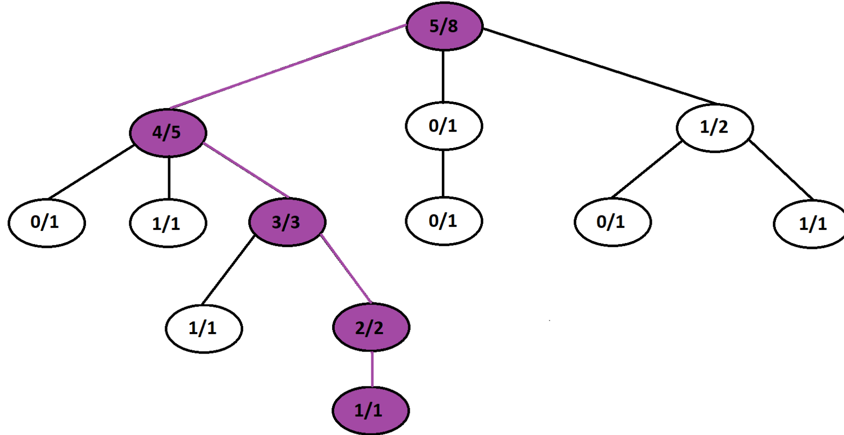

Figure 4: MCTS backpropagation phase

3.3 Upper Confidence Bounds Applied To Trees

One of the major challenges of MCTS lies in the selection of child nodes, also called selection policy. The difficulty stems from balancing the exploitation of known good actions and exploration of less visited nodes to find potentially better actions. The most prominent selection policy in use is the Upper Confidence Bound 1 applied to Trees (UCT) [14]. It advises to award each child node a score according to the following formula:

$$\frac{w_i}{n_i} + c * \sqrt{\frac{lnN_i}{n_i}}$$

w_i: win count of child node i
n_i: visit count of child node i
c: exploration constant
N_i: visit count of parent node i

The expression consists of two parts. The left part of the expression represents the exploitation value, and the right part represents the exploration value. The former is merely the rate at which the child node has won its simulations, while the latter consists of the ratio between how often the parent node and child node have been visited, multiplied by an exploration constant. There is no optimal exploration constant across all games, since different games require different balancing

of exploitation versus exploration, it needs to be chosen empirically. Generally said, the more simulations a node wins, the higher its exploitation part. Likewise, the less explored the node is, the higher its exploration part. The combined parts determine the score a child node obtains during the selection, and the child with the highest score is chosen.

3.4 Parallelization

There have been many studies conducted regarding the parallelization of the algorithm in order to increase the number of iterations per second. It has to be noted that in contrast to the traditional use of parallelization, where the sequential version and the parallelized version must provide the same results, the resulting search trees of the sequential and parallelized versions are not the same. Thus, it could be argued that these parallelization efforts are simply variants of MCTS. Nonetheless, in this thesis they will be referred as parallelized MCTS versions or approaches. In 2007 and 2008 three key approaches have been proposed:

- Leaf parallelization [4] is the simplest parallelization method, where only the simulation phase is being executed in parallel. Because of this, it is fairly simple to implement, however, the gains are minimal as well. The advantage of this approach is the algorithm is able to gather the statistics of multiple simulations from the expanded node in a single iteration, instead of just one simulation. The downside is that the search tree does not grow any faster, on the contrary, it may even grow slower due to the fact that all threads have to wait for the slowest thread to finish its simulation and backpropagate its results before the next iteration begins.

- Tree parallelization [5] executes all four phases in parallel. All threads share the same search tree and use mutexes or locks to protect nodes from simultaneous write operations. Because all four phases are parallelized, the search tree grows considerably faster compared to the sequential version or leaf parallelization. Regardless, it is also more difficult to implement, since additional effort has to be made to ensure proper synchronization.

- Root parallelization [6] differs from the previous two methods by having each thread build their own search tree. The threads have their own local copies of the same root node and, once the time limit has been reached, their local statistics are aggregated. The major advantage of this method is that, except when the individual statistics are being collected after the time is up, no synchronization is needed since each thread constructs their own search tree. On the other hand, having multiple search trees with duplicate child nodes may lead to problematic memory consumption.

Each of these methods has been subsequently employed in the exploration of more sophisticated parallelization methods. For example, by combining root and tree parallelization Swiechowski et al. [25] proposed root-tree parallelization, or, Mirsoleimani et al. [18] presented a lock-free variation of tree parallelization. In both examples the aims were to improve the scalability of parallelized MCTS, which root and tree parallelization both lack.

3.5 Vienna Game AI Library

All contributions of this thesis are in the Vienna Game AI library (VGAI), which is a single-header C++ library that features a collection of functionalities for game development, developed by Lavinia-Elena Lehaci [15] as part of her master's thesis. It is primarily aimed at assisting game developers, who benefit from not having to implement algorithms, such as A* pathfinding or flocking behaviour, from scratch and are instead able to integrate pre-existing implementations easily into their projects. It is hosted on GitHub, and, building and compiling is handled by CMake

4 Implementing MCTS

4.1 Implementation Design

In the design process of the implementation of the algorithm two goals were being pursued simultaneously:

1. Game developers using the library should only need to implement as little functionality as possible to be able to integrate the algorithm into their projects.

2. The algorithm should be able to deliver a satisfying performance without requiring the incorporation of domain knowledge.

To achieve this, the core logic of the algorithm was broken down into four concrete classes. Developers need to extend two of these classes by overriding one method in each and integrate their own game logic into them in order for the algorithm to work.

4.2 Class Action

The class Action is the first class that is of interest for game developers. It does not contain any logic in of itself, as can be seen in Listing 1, with the only relevant method being the virtual void execute(DS& gameState) method. Developers using the library should extend the class and override the method, and add the execution

of the action onto the referenced game state. By using templation on the passed game state object the action can be used on any type of derived game state class.

```
template<typename DS>
    class Action {
    public:
        Action() {}

        virtual void execute(DS& gameState) {};
    };
```

Listing 1: class Action

4.3 Class MCTSState

```
template<typename DS, typename DA>
    class MCTSState {
    protected:
        std::vector<DA> actions;
        bool isTerminal = false;
        std::string playerTurnId = ""; // Whose player's turn it is
     based on their id
        std::string winnerPlayerId = ""; // The id of the winning
   player

    public:
       //Constructors, Setter & Getter
       ...

        virtual bool getIsTerminal() {
            return isTerminal;
        }

        ...
    };
```

Listing 2: class MCTSState excerpt

The second class that needs to be worked with is the class MCTSState. This class represents the current game state in a game. Relevant of the class are shown in Listing 2. Unlike the class Action, this class has four member variables that need to be worked with to ensure the algorithm can function properly: a vector consisting of all possible actions that can be taken in the current game state, a check if the game is in a terminal state, whose player's turn it is and, if the game has concluded, which player has won the game. Setter and Getter methods to manipulate these member variables are provided and do not neet to be modified. However, the virtual bool getIsTerminal() method needs to be overridden as such that it evaluates whether

a winning condition has been met. Although it may seem redundant to have this method if the developers decide to change the isTerminal variable in the overridden execute(DS& gameState) method of class Action, it was a deliberate design decision to separate these concerns in order to lower interdependency.

4.4 Class MCTSNode

The only class game developers do not have any direction interaction with is the class MCTSNode. An instance of this class represents a single node in the search tree. As shown in Listing 3, a MCTSNode object has information about its child nodes and its parent node, the win and visit counts, and the action is took to get from the parent node to this node. It also contains one mutex per instance to ensure locking for synchronisation purposes when the algorithm uses multi-threading. For the algorithm itself two methods are of particular importance. The first being the bool isCompletelyExpanded() method which is a simple check if the size of the action set equals the size of the vector filled with its child nodes. The second is the double getUCT(...) const method that calculates the UCT value for this node. Since the algorithm should be applicable to a broad range of games, the exploitation part of the formula is inverted if the id of the player who had to take an action last turn and the id of the player who has to take an action in the current turn do not match. As a result, if a player has to take consecutive turns, their seemingly better performing actions are rewarded. On the other hand, if in the subsequent turn the enemy player has to perform an action the inversion exploits actions that result in the enemy performing worse.

```
1  template<typename DS, typename DA>
2      class MCTSNode {
3      private:
4          DS state;
5          std::vector<std::shared_ptr<MCTSNode<DS, DA>>> children =
   {};
6          std::weak_ptr<MCTSNode<DS, DA>> parent;
7          int num_visits = 0;
8          int num_wins = 0;
9          DA actionToGetHere;
10
11     public:
12         std::mutex node_mutex;
13
14         bool isCompletelyExpanded() {
15             return children.size() == state.getActions().size();
16         }
17
18         double getUCT(const double& c_value, const std::string&
   rootPlayerId) const {
```

```
19              double exploit;
20              double explore;
21              double UCT;
22              if (num_visits == 0) {
23                  return std::numeric_limits<double>::infinity();
24              }
25
26              exploit = getWinrate();
27
28              if (rootPlayerId != state.getPlayerTurnId()) {
29                  exploit = 1 - getWinrate();
30              }
31
32              explore = c_value * std::sqrt(std::log(static_cast<
    double>(parent.lock()->getVisits())) / static_cast<double>(
    num_visits));
33              UCT = exploit + explore;
34
35              return UCT;
36          }
37
38          ...
39      };
```

Listing 3: class MCTSNode excerpt

4.5 Class MCTS

The main logic of the algorithm is contained in the class MCTS. This class does not have any member variables and has only one public method, as can be seen in Listing 4. The method DA runMCTS(...) takes three input parameters, the current game state, the time limit for the algorithm and the exploration value, as well as one optional parameter to set the thread count if the algorithm should make use of parallelization. A root node is constructed and passed as reference to every thread which, in the method searchBestAction(...), expand the same search tree through tree parallelization. Once the threads join the action with of the most visited child node is returned. With the use of std::shared_pointer, memory is automatically freed once the algorithm concludes and the objects the std::shared_pointers point to go out of scope. One reason why tree parallelization was chosen over root parallelization was to omit the need for an action id as an identifier across the different search trees and instead be able to reduce the input needed by users. Another aspect is that in their research, Chaslot et al. [6] find that root parallelization and tree parallelization deliver a similar playing strength in various board games up until 16 threads, which is more than are currently being used in computer games.

```cpp
 DA runMCTS(DS currentState, const double& timeLimitMs, double c
, int thread_count = 1) {

     std::vector<std::jthread> thread_vector;
     auto root = std::make_shared<MCTSNode<DS, DA>>(currentState
, std::weak_ptr<MCTSNode<DS, DA>>());

     for (int i = 0; i < thread_count; i++) {
         thread_vector.emplace_back([&]() {
             searchBestAction(root, timeLimitMs, c);
             });
     }

     for (auto& t : thread_vector) {
         if (t.joinable()) {
             t.join();
         }
     }

     // Get best child
     std::shared_ptr<MCTSNode<DS, DA>> bestChild = nullptr;
     for (auto& child : root->getChildren()) {
         if (bestChild == nullptr || child->getVisits() >
bestChild->getVisits()) {
             bestChild = child;
         }
     }

     return bestChild->getAction();
 }
```

Listing 4: runMCTS method

The respective methods of each phase of the algorithm are called iteratively in the searchBestAction(...) method until the time limit has been reached, which is determined after every iteration in order to not inefficiently assess the remaining search time.

Listing 5 shows the selection process where the current node a thread is at is locked and then checked if said node is either not yet fully expanded or terminal, in which case the selection stops at that node.

```cpp
std::shared_ptr<MCTSNode<DS, DA>> select(std::shared_ptr<MCTSNode<
DS, DA>>& currentNode, double& c_value, std::string&
rootPlayerId) {
 std::lock_guard<std::mutex> lock(currentNode->node_mutex);
 // If at least one action not tried or game over don't go
deeper
 if (currentNode->isCompletelyExpanded() == false || currentNode
->getState().getIsTerminal() == true) {
```

```
5         return currentNode;
6     }
7     std::shared_ptr<MCTSNode<DS, DA>> bestChild = nullptr;
8 
9     double bestUCT_value = std::numeric_limits<double>::lowest();
10 
11     // Get best child according to UCT formula
12     for (auto& child : currentNode->getChildren()) {
13         std::lock_guard<std::mutex> lock(child->node_mutex);
14         double child_UCT = child->getUCT(c_value, rootPlayerId);
15 
16         if (child_UCT > bestUCT_value) {
17             bestUCT_value = child_UCT;
18             bestChild = child;
19         }
20     }
21 
22     return bestChild;
23 }
```

Listing 5: Selection phase implementation

Once a node has been selected the expansion phase occurs, as shown in Listing 6, where the expand method expands a node with a randomly selected action from its action set executed if, and only if, the node is not in a terminal state and still has nodes that can be expanded, otherwise this phase is skipped.

```
1 std::shared_ptr<MCTSNode<DS, DA>> expand(std::shared_ptr<MCTSNode<
   DS, DA>>& currentNode) {
2     std::lock_guard<std::mutex> lock(currentNode->node_mutex);
3 
4     // If game is over or all actions have been tried don't create
   new node
5     if (currentNode->getState().getIsTerminal() || currentNode->
   isCompletelyExpanded()) {
6         return currentNode;
7     }
8 
9     if (currentNode->getState().getUntriedActions().empty())
10     {
11         currentNode->getState().setUntriedActions();
12     }
13 
14     // Get an action that has not been tried yet and is next in
   line and create new node
15     thread_local std::mt19937 rng(std::random_device{}());
16     std::uniform_int_distribution<int> dist(0, currentNode->
   getState().getUntriedActions().size() - 1);
17     int index = dist(rng);
```

```
18     auto randomAction = currentNode->getState().getUntriedActions()
    [index];
19     currentNode->getState().removeTriedAction(index);
20 
21     DS stateToAdd(currentNode->getState());
22     randomAction.execute(stateToAdd);
23     auto child = std::make_shared<MCTSNode<DS, DA>>(stateToAdd,
    currentNode, randomAction);
24     currentNode->addChild(child);
25 
26     return child;
27 }
```

Listing 6: Expansion phase implementation

If no new node could be expanded then the simulation method performs a rollout at the selected node during the selection phase, otherwise the newly expanded node is chosen. Here random actions are taken until the game is over, as can be seen in Listing 7.

```
1 void simulate(std::shared_ptr<MCTSNode<DS, DA>>& currentNode) {
2     auto currentState(currentNode->getState());
3 
4     // While not game over take random actions to play until game
    is over
5     thread_local std::mt19937 rng(std::random_device{}());
6 
7     while (currentState.getIsTerminal() == false) {
8         auto actions = currentState.getActions();
9         if (actions.empty()) {
10             break;
11         }
12         std::uniform_int_distribution<int> distr(0, actions.size()
   - 1);
13         int index = distr(rng);
14 
15         try {
16             auto randomAction = actions[index];
17             randomAction.execute(currentState);
18         }
19         catch (const std::exception& e) {
20             std::cerr << "Exception: " << e.what() << '\n';
21             break;
22         }
23     }
24 
25     backpropagate(currentNode, currentState.getWinnerPlayerId());
26 }
```

Listing 7: Simulation phase implementation

In Listing 8 result of the simulation phase is backpropagated up until the root node is reached, with locking the current node during the incrementations of the win count, if the player id of the winner matched the one whose turn it is, and visit count to handle race conditions.

```
1 void backpropagate(std::shared_ptr<MCTSNode<DS, DA>>& currentNode,
    const std::string& winnerPlayerId) {
2     auto backpropagatedNode = currentNode;
3     // As long as not root increase visits upwards and if winner is
      the same as current player of each node increase wins as well
4     while (backpropagatedNode != nullptr) {
5         {
6             std::lock_guard<std::mutex> lock(backpropagatedNode->
    node_mutex);
7             backpropagatedNode->incrementVisits();
8
9             if (winnerPlayerId == backpropagatedNode->getState().
    getPlayerTurnId()) {
10                backpropagatedNode->incrementWins();
11            }
12        }
13
14        backpropagatedNode = backpropagatedNode->getParent();
15    }
16 }
```

Listing 8: Backpropagation phase simulation

4.6 Examplatory Use

The incorporation of the library is fairly simple and straightforward. In order to use the algorithm the ViennaGameAILibrary.hpp file needs to be included as a header. Aftwards both class Action and class MCTSState need to be extended and incorporated into the game. Listing 8 shows the addition of the library into a simple Connect Four game, which will be used in the subsequent section as part of the evaluation. The class ConnectFourGameState extends the class MCTSState and overrides the getIsTerminal() method by incorporating a validation if its own win conditions have been met.

```
1 class ConnectFourGameState : public MCTSState<ConnectFourGameState,
     ConnectFourAction> {
2 private:
3     std::vector<Cell> gameBoard = {};
4     using MCTSState<ConnectFourGameState, ConnectFourAction>::
    actions;
5     using MCTSState<ConnectFourGameState, ConnectFourAction>::
    isTerminal;
```

```
    using MCTSState<ConnectFourGameState, ConnectFourAction>::playerTurnId;
    using MCTSState<ConnectFourGameState, ConnectFourAction>::winnerPlayerId;

public:
    bool getIsTerminal() override {
        if (actions.size() == 0) {
            return true;
        }
        std::vector<Cell> filledCells;
        std::for_each(gameBoard.begin(), gameBoard.end(), [&](auto& cell) { if (cell.getPlayerChip() != "0") { filledCells.push_back(cell); }});

        std::string checkWin = "";
        for (const auto& cell : filledCells) {
            checkWin = getHorizontalWinChip(cell);
            if (checkWin != "") {
                setWinnerPlayerId(checkWin);
                return true;
            }

            checkWin = getVerticalWinChip(cell);
            if (checkWin != "") {
                setWinnerPlayerId(checkWin);
                return true;
            }

            checkWin = getDiagonalWinChip(cell);
            if (checkWin != "") {
                setWinnerPlayerId(checkWin);
                return true;
            }
        }

        return false;
    }

    ...
};
```

Listing 9: class ConnectFourGameState

The class ConnectFourAction extends class Action and simply sets the value of one cell according to the player's id, then sets the opponent's id as the next player's turn and generates its possible action set.

```
class ConnectFourAction : public Action<ConnectFourGameState> {
...
```

```
3
4 public:
5     void execute(ConnectFourGameState& gameState) override;
6
7 ...
8 };
9
10 void ConnectFourAction::execute(ConnectFourGameState& gameState) {
11     for (auto& entries : gameState.getGameBoard()) {
12         if (entries.getX() == x && entries.getY() == y) {
13             entries.setPlayerChip(playerId);
14             if (gameState.getPlayerTurnId() == "1") {
15                 gameState.setPlayerTurnId("2");
16             }
17             else {
18                 gameState.setPlayerTurnId("1");
19             }
20         }
21     }
22
23     gameState.generateActions();
24 }
```

Listing 10: class ConnectFourAction

5 Evaluation

This section presents the results of testing our implementation in three games with different complexities, namely Tic-Tac-Toe, Connect Four and Hex. The goal of the evaluation is to analyze both the playing strength of the algorithm in relation to an increase in the time limit being given, and, assess the performance differences across different thread counts. For each configuration a test run of 300 games was conducted, with time limits ranging between 10 ms and 100 ms, and an exploration constant of 2 was being used. The tests were being run on the following hardware:

Component	Model
CPU	AMD Ryzen 3 3100
GPU	Nvidia GTX1650 Super
RAM	Corsair 2x8GB DDR4

Table 1: Testing hardware

5.1 Experiment Results

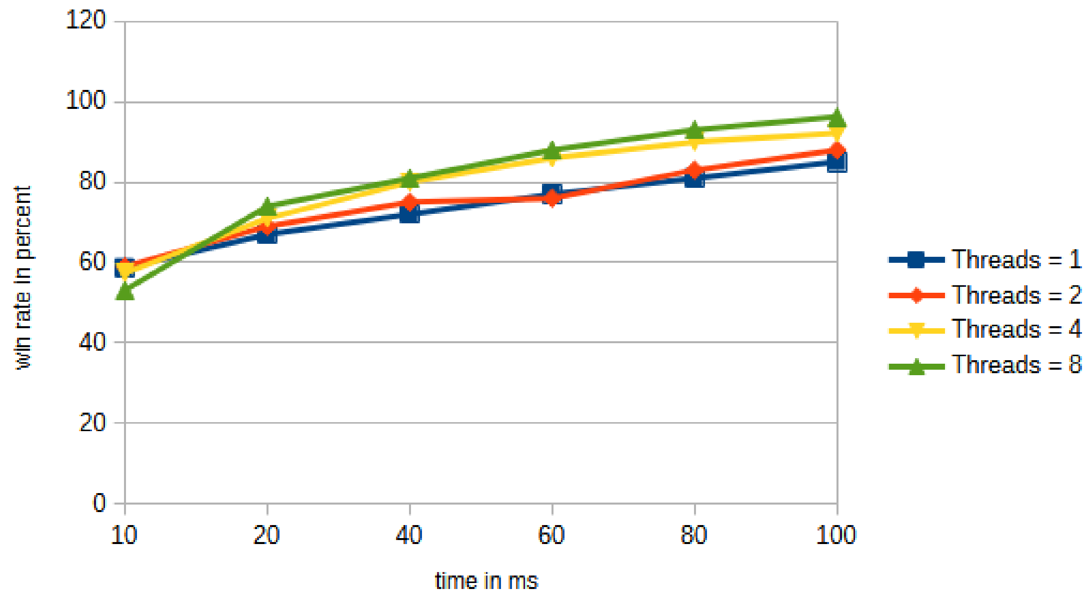

Figure 5: MCTS win rates in Tic-Tac-Toe

Looking at the results in Figure 5 we can conclude that while there is a significant increase in the decision-making quality of the algorithm in Tic-Tac-Toe when the time limit is being increased, the relative difference in performance between the different thread counts remains unchanged in the lower-end and higher-end time limits, with only a slight, temporary increase forming between 40 ms and 60 ms.

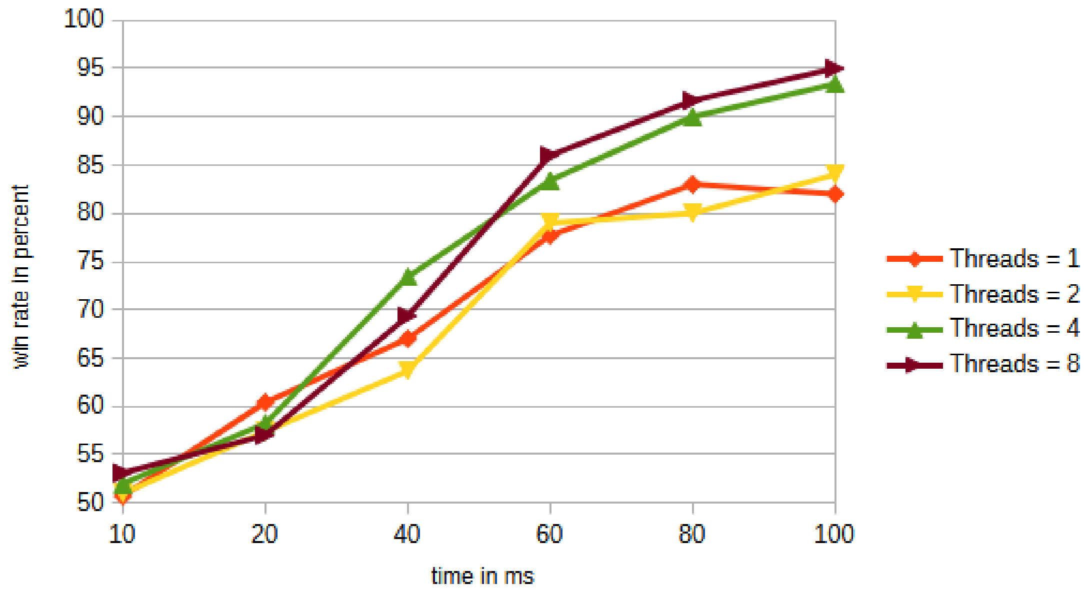

Figure 6: MCTS win rates in Connect Four

Figure 6 shows the performance of the implementation in a simple 7x6 Connect Four game. It can be seen that there is not a noticeable difference in playing strength between the algorithm running one and two threads, with the latter being barely superior at a time limit of 100 ms. However, there is an observable jump in performance when the algorithm runs on either four or eight threads, with an increase in time limit widening the gap. Most notably, across all thread counts the algorithm reaches diminishing returns at around a time limit of 60 ms. Additionally, at a time limit of 100 ms the lower thread counts reach win rates in the lower 80%-range, while the higher thread counts reach win rates in the mid 90%-range.

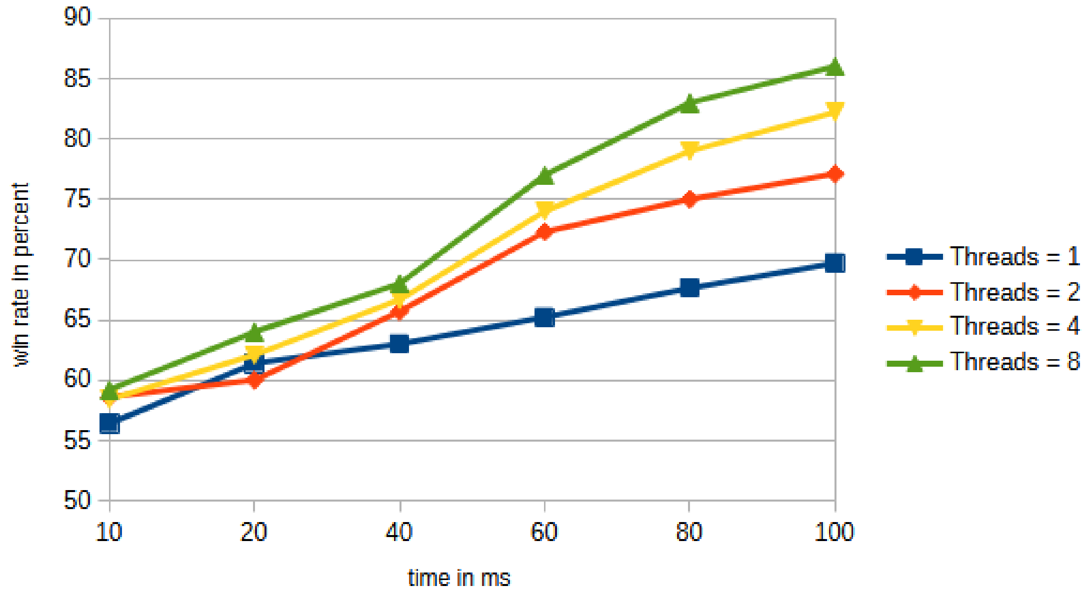

Figure 7: MCTS win rates in Hex

In Figure 7 the playing strength of the algorithm in Hex shows a greater divergence across the different thread counts with increasing time limits than in the previous two figures. We can also conclude that the algorithm still makes noticeable increases in win rates over all thread counts in the upper end of the time limits, indicating that a further increase in the time limit beyond 100 ms may yield an even stronger performance, even though the performance improvements have started to slowly decline.

5.2 Discussion

Based on the results of the different configurations in our test games we can now answer our research question:

- How does the playing strength of MCTS without domain knowledge scale with the number of threads across games with different complexities?

Our findings demonstrate that in games with low branching factors, such as Tic-Tac-Toe with an initial branching factor of 9 that is being decreased the more the game has been played, a scaling thread count only leads to a marginal increase in performance difference. In games with higher, yet still considered relatively low, branching factors like Connect Four, which has a consistent branching factor of $\tilde{7}$,

a noticeable discrepancy can be observed between lower and higher thread counts. However, no discrepancy can be seen across the lower thread counts and across the higher thread counts. Finally, Hex, which has an initial branching factor of $\tilde{1}21$ in the 11x11 version that was tested on, shows diverging performance differences as more search time is being assigned to the algorithm. We can conclude that in games with higher branching factors a scaling thread count leads to an amplified increase in performance. One way to explain these findings is that in domains with greater search spaces the algorithm can make better use of multi-threading, whereas in domains with small search spaces a lot of computational power is being wasted due to the individual threads waiting a lot more for locked nodes to be unlocked, especially if the algorithm is stuck in a local optimum.

6 Conclusion

MCTS has shown to be a well-performing decision-making algorithm, even without the use of heuristics, across a broad range of games. Combined with a strong selection policy like UCT and the incorporation of reinforced learning the algorithm has been able to be applied in domains beyond games. This thesis, however, remained focused on the use of MCTS in game development and presented an MCTS implementation for the VGAI library for game developers to use. The aim hereby was to take as much implementation overhead off of the developers, with them having only to implement two methods themselves to integrate the algorithm into their projects. To illustrate the playing strength of the algorithm tests have been run on three simple turn-based computer games: Tic-Tac-Toe, Connect Four and Hex. The games have been used to evaluate the performance of the algorithm in different configurations, like single- and multi-threading. The findings have been analysed to assess the scaling of parallelized MCTS in games across different complexities.

Potential future work could lie in the extension of the algorithm to be able to incorporate heuristics to deliver an even better performance. With this addition the algorithm would also be more attractive in the RTS game domain, even though in theory playing strength the implementation in RTS games would depend on how large the search space is, given the more restricting time limits. Another aspect could be the addition of either progressive widening or double progressive widening to ensure better exploration in extremely large action or search spaces.

References

[1] Abramson, B. *The Expected-Outcome Model of Two-Player Games*. PhD thesis, Columbia University, New York, NY, USA, 1987. Doctoral dissertation.

[2] An, B., Ordóñez, F., Tambe, M., Shieh, E., Yang, R., Baldwin, C., DiRenzo, J., Moretti, K., Maule, B., and Meyer, G. A deployed quantal response-based patrol planning system for the u.s. coast guard. *Interfaces 43*, 5 (2013), 400–420.

[3] Best, G., Cliff, O., Patten, T., Mettu, R., and Fitch, R. Dec-mcts: Decentralized planning for multi-robot active perception. *The International Journal of Robotics Research 38* (03 2018), 027836491875592.

[4] Cazenave, T., and Jouandeau, N. On the parallelization of uct.

[5] Cazenave, T., and Jouandeau, N. A parallel monte-carlo tree search algorithm. In *Computers and Games* (Berlin, Heidelberg, 2008), H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., Springer Berlin Heidelberg, pp. 72–80.

[6] Chaslot, G. M. J. B., Winands, M. H. M., and van den Herik, H. J. Parallel monte-carlo tree search. In *Computers and Games* (Berlin, Heidelberg, 2008), H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., Springer Berlin Heidelberg, pp. 60–71.

[7] Couëtoux, A., Hoock, J.-B., Sokolovska, N., Teytaud, O., and Bonnard, N. Continuous upper confidence trees. In *Learning and Intelligent Optimization* (Berlin, Heidelberg, 2011), C. A. C. Coello, Ed., Springer Berlin Heidelberg, pp. 433–445.

[8] Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games* (Berlin, Heidelberg, 2007), H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, Eds., Springer Berlin Heidelberg, pp. 72–83.

[9] Cowling, P. I., Powley, E. J., and Whitehouse, D. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games 4*, 2 (2012), 120–143.

[10] Fang, F., Stone, P., and Tambe, M. When security games go green: Designing defender strategies to prevent poaching and illegal fishing. In *International Joint Conference on Artificial Intelligence* (2015).

[11] Gelly, S. Combining online and offline knowledge in uct. *ACM International Conference Proceeding Series 227* (06 2007).

[12] Hennes, D., and Izzo, D. Interplanetary trajectory planning with monte carlo tree search. In *Proceedings of the 24th International Conference on Artificial Intelligence* (2015), IJCAI'15, AAAI Press, p. 769–775.

[13] Jain, M., Tsai, J., Pita, J., Kiekintveld, C., Rathi, S., Tambe, M., and Ordóñez, F. Software assistants for randomized patrol planning for the lax airport police and the federal air marshal service. *Interfaces 40*, 4 (2010), 267–290.

[14] Kocsis, L., and Szepesvári, C. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006* (Berlin, Heidelberg, 2006), J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., Springer Berlin Heidelberg, pp. 282–293.

[15] Lehaci, L.-E. Vienna game ai library. Master's thesis, University of Vienna, Vienna, Austria, 2024. Master's thesis.

[16] Lubosch, M., Kunath, M., and Winkler, H. Industrial scheduling with monte carlo tree search and machine learning. *Procedia CIRP 72* (2018), 1283–1287. 51st CIRP Conference on Manufacturing Systems.

[17] Metropolis, N., and Ulam, S. The monte carlo method. *Journal of the American Statistical Association 44*, 247 (1949), 335–341. PMID: 18139350.

[18] Mirsoleimani, S. A., van den Herik, J., Plaat, A., and Vermaseren, J. A lock-free algorithm for parallel mcts. In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART,* (2018), INSTICC, SciTePress, pp. 589–598.

[19] Neto, T., Constantino, M., Martins, I., and Pedroso, J. P. A multi-objective monte carlo tree search for forest harvest scheduling. *European Journal of Operational Research 282* (09 2019).

[20] Ontanon, S. The combinatorial multi-armed bandit problem and its application to real-time strategy games. *Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2013 9* (10 2013), 58–64.

[21] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., and Lillicrap, T. Mastering atari, go, chess and shogi by planning with a learned model. *Nature 588* (12 2020), 604–609.

[22] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature 529* (2016), 484–503.

[23] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science 362*, 6419 (2018), 1140–1144.

[24] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T. P., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature 550* (2017), 354–359.

[25] Swiechowski, M., and Mandziuk, J. A hybrid approach to parallelization of monte carlo tree search in general game playing. In *Challenging Problems and Solutions in Intelligent Systems*, G. D. Tré, P. Grzegorzewski, J. Kacprzyk, J. W. Owsinski, W. Penczek, and S. Zadrozny, Eds., vol. 634 of *Studies in Computational Intelligence*. Springer, 2016, pp. 199–215.

[26] Tomy, M., Seiler, K., and Hill, A. Mcts based dispatch of autonomous vehicles under operational constraints for continuous transportation, 07 2024.

[27] Uriarte, A., and Ontanon, S. Single believe state generation for handling partial observability with mcts in starcraft. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 13*, 2 (Oct. 2017), 15–20.

[28] Wijaya, T. K., Papaioannou, T. G., Liu, X., and Aberer, K. Effective consumption scheduling for demand-side management in the smart grid using non-uniform participation rate. In *Sustainable Internet and ICT for Sustainability, SustainIT 2013, Palermo, Italy, 30-31 October, 2013, Sponsored by the IFIP TC6 WG 6.3 "Performance of Communication Systems"* (2013), IEEE Computer Society, pp. 1–8.